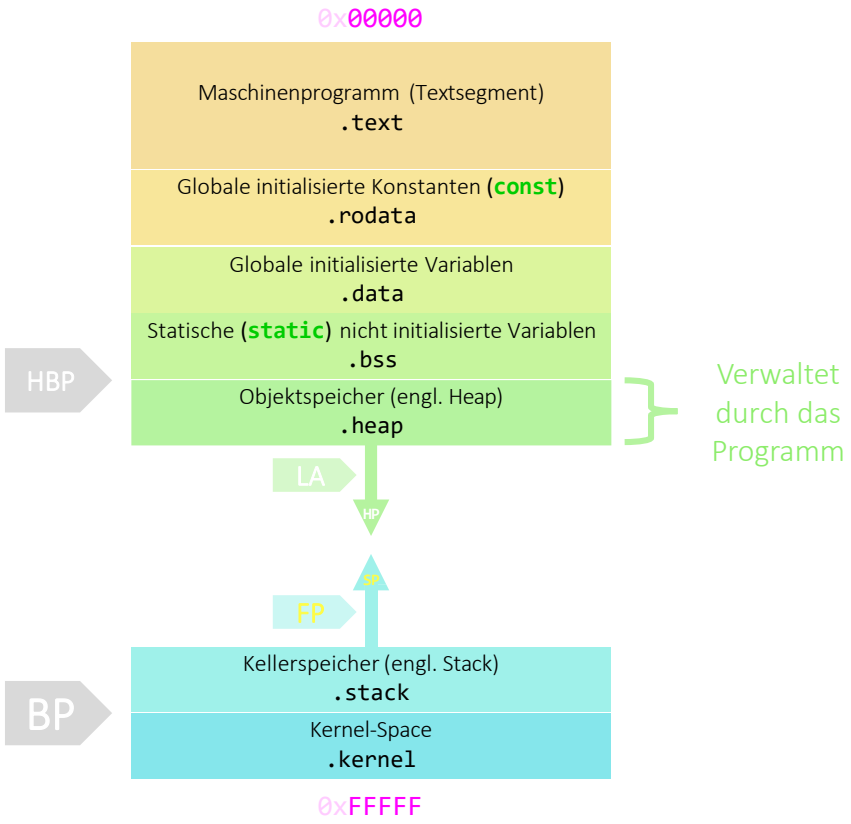
# Speicherverwaltung

Emilio Pielsticker

# Speicherlayout eines Prozesses




HBP ≡ heap
LA ≡ last_alloc_chunk_idx

# Dynamische Speicherverwaltung in C

- Funktion zur Allokation von Speicher: `malloc(size_t size)`
- Reserviert zur Laufzeit Speicher auf dem Heap
- `size` legt die Größe des zu reservierenden Bereichs in Bytes fest
- Rückgabewerte:
  - ≠ `NULL`: wenn Allokation erfolgreich
  - = `NULL`: wenn ein Fehler aufgetreten ist

# Initialisierung mit Null

- `void* calloc(size_t` $n$`, size_t` $m$`)`
  - Reserviert für $n$ Objekte der Größe $m$ im Heap $n \cdot m$-Bytes und initialisiert diesen mit $0$
  - Bei `malloc` ist dies allgemein nicht der Fall:
    Reservierter Speicher kann möglicherweise noch Altlasten beinhalten!

# Speicherplatz freigeben

- Gegenstück zu `malloc` (bzw. `calloc`) ist `free(void* ptr)`
- Gibt Speicher an Adresse `ptr` wieder frei
- Speicher darf nur einmal freigegeben werden
- `free` bedarf keine Fehlerbehandlung

# Array im Heap

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr;
    ptr = malloc(n * sizeof(*ptr));
    if (ptr == NULL) { perror("malloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    return 0;
}
```

# Initialisierung mit Null

- `void* calloc(size_t` $n$`, size_t` $m$`)`
  - Reserviert für $n$ Objekte der Größe $m$ im Heap $n \cdot m$-Bytes und initialisiert diesen mit $0$
  - Bei `malloc` ist dies allgemein nicht der Fall:
    Reservierter Speicher kann möglicherweise noch Altlasten vom Stack oder vom Heap beinhalten!

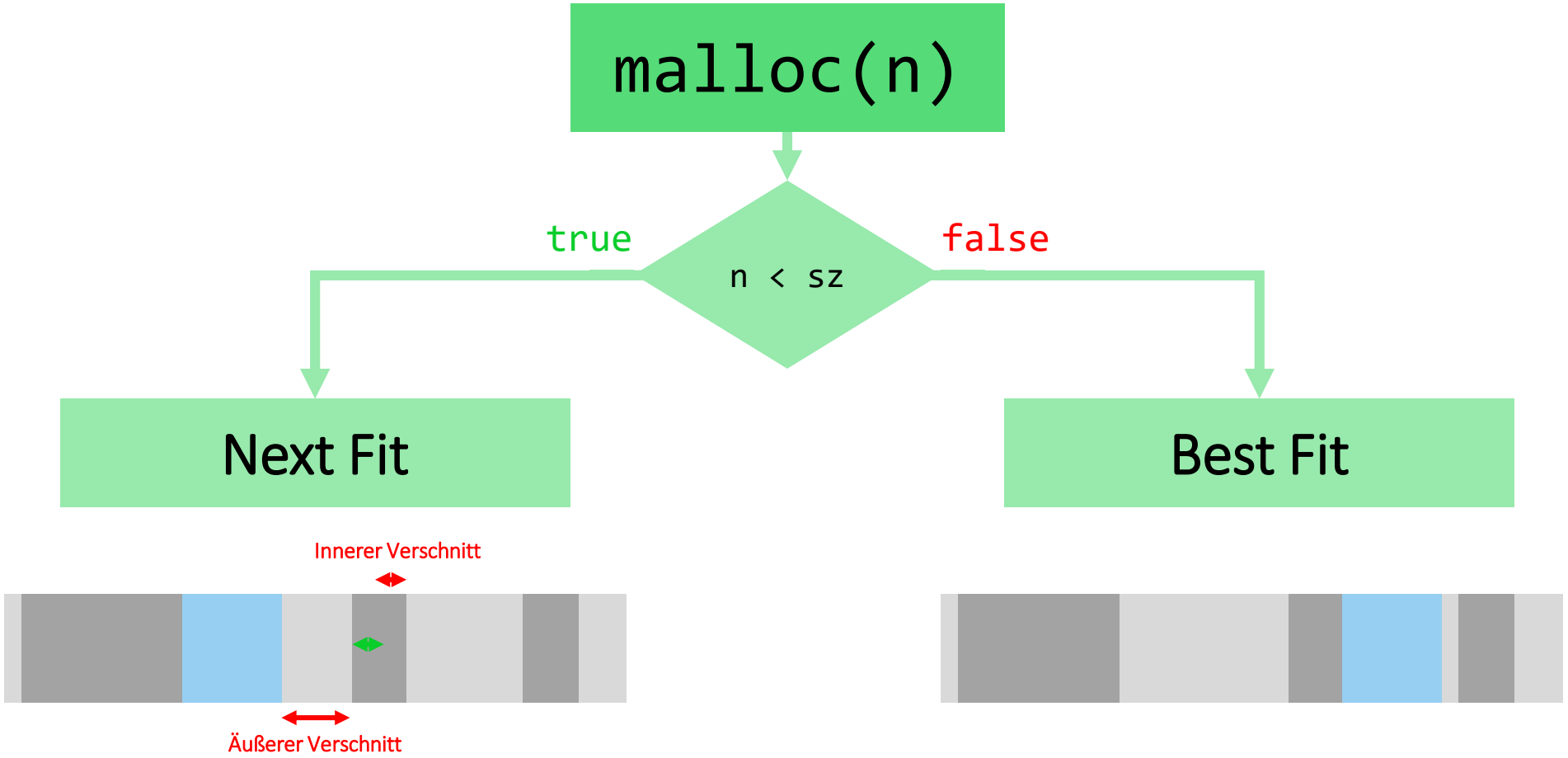
# Implementierung von `calloc`

```
void* mycalloc(size_t nmemb, size_t size) {
    size_t total_size = nmemb * size;
    void* ptr = mymalloc(total_size);
    if (ptr) { memset(ptr, 0, total_size); }
    return ptr;
}
```

# Array im Heap mit `calloc`

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr = (int*)(calloc(n, sizeof(*int)));
    if (ptr == NULL) { perror("calloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    return 0;
}
```

# Platzierungsstrategien



# Speicherplatzierungsstrategie 'Next Fit'

- Sucht nächste passende Lücke im Speicher
- Merkt sich Position der letzten Allokation, setzt Suche bei nächster Anfrage dort fort
- Schnelle Platzierungsstrategie die sich besonders für sehr kleine Blöcke gut eignet

HBP ≡ `heap`
LA ≡ `last_alloc_chunk_idx`

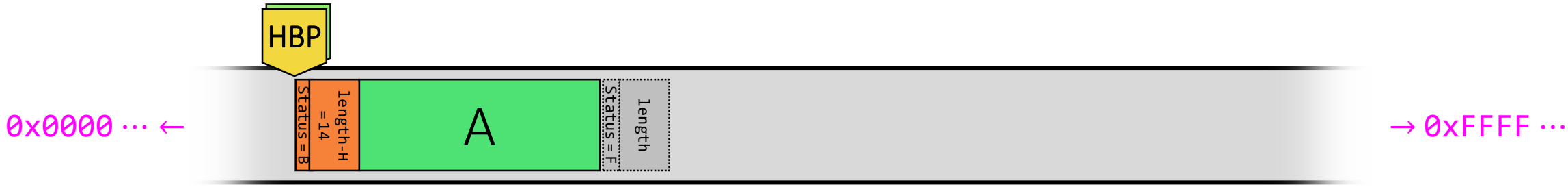
# Allokation

- Metadaten pro Block
  - Status des Blocks: **frei** oder **belegt**
  - Länge des Blocks: falls $0$, ist der Status irrelevant
- Anfänglich können Daten hintereinander in den Speicher gelegt werden

# Ausgangszustand

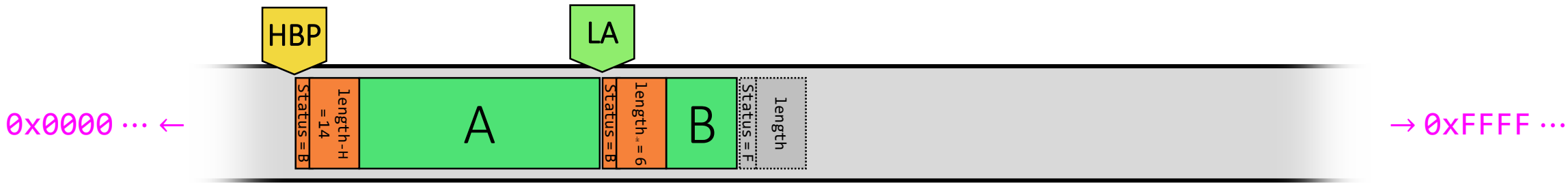

**H** entspricht hier der Länge der Metadaten.

# Allokation eines Blockes $A$ der Länge $14$



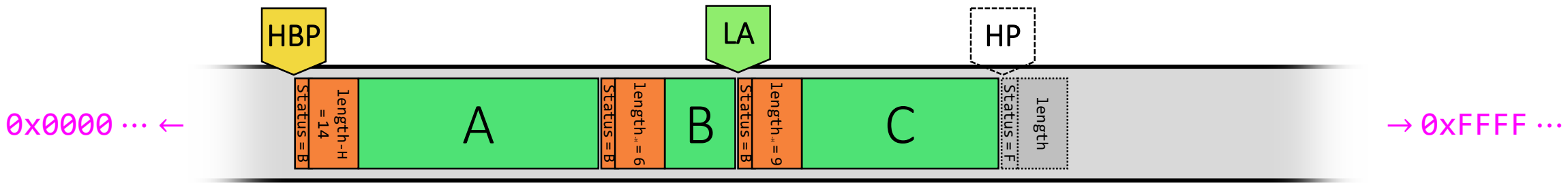

**H** entspricht hier der Länge der Metadaten.

# Allokation eines Blockes $B$ der Länge $6$



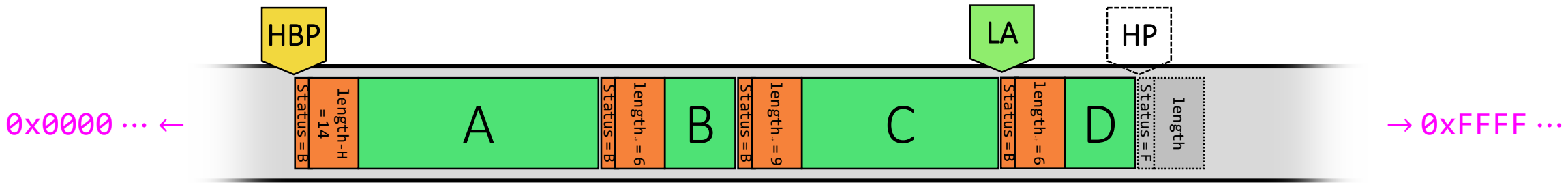

**H** entspricht hier der Länge der Metadaten.

# Allokation eines Blockes C der Länge 9




H entspricht hier der Länge der Metadaten.

# Allokation eines Blockes $D$ der Länge $6$




**H** entspricht hier der Länge der Metadaten.

# Reallokation

Aktualisieren des Zeigers

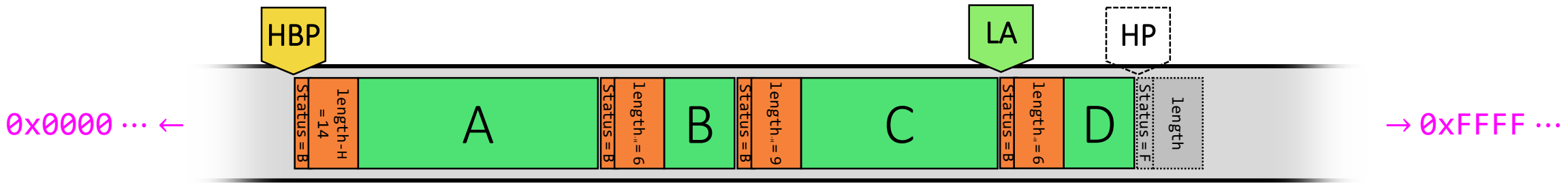
- `ptr = realloc(ptr, n)` ändert die Größe des Blocks `*ptr`
- Gelingt `realloc`, besitzt der Block `*ptr` die Größe $n$
- Schlägt `realloc` fehl, wird `NULL` zurückgeliefert

- `realloc(ptr, 0)` entspricht einem `free(ptr)`
- `realloc(NULL, n)` entspricht einem `malloc(n)`

# Reallokation im Beispiel

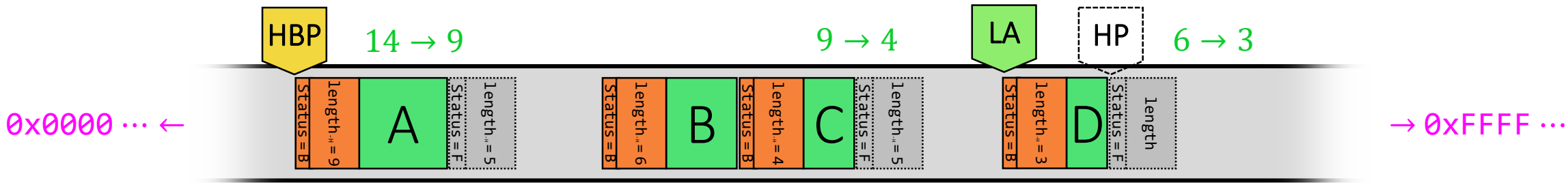
- Blöcke können **in-place** realloziert werden,
  indem der Status entsprechend angepasst wird
- Vergrößern ist so nur möglich, wenn hinter
  dem Block eine entsprechend große Lücke frei ist
- Ansonsten müsste ein weiterer Block mit der
  geforderten Größe reserviert werden,
  in den der Inhalt dann hineinkopiert wird
  → Den Originalblock würde man danach freigeben

# Blöcke vor einer in-place Reallokation




**H** entspricht hier der Länge der Metadaten.

# Blöcke nach der in-place Reallokation

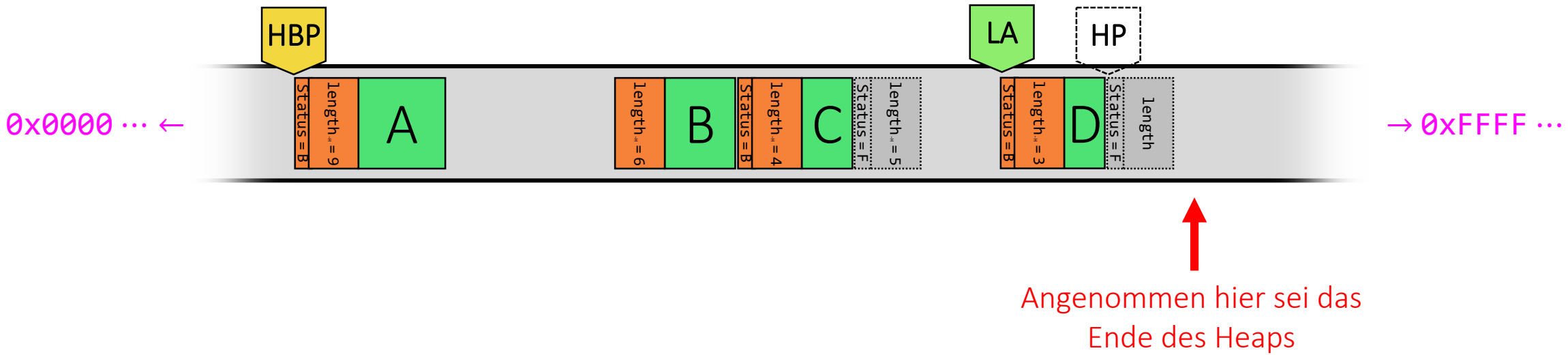



**H** entspricht hier der Länge der Metadaten.

# Suche nach entsprechend großer Lücke

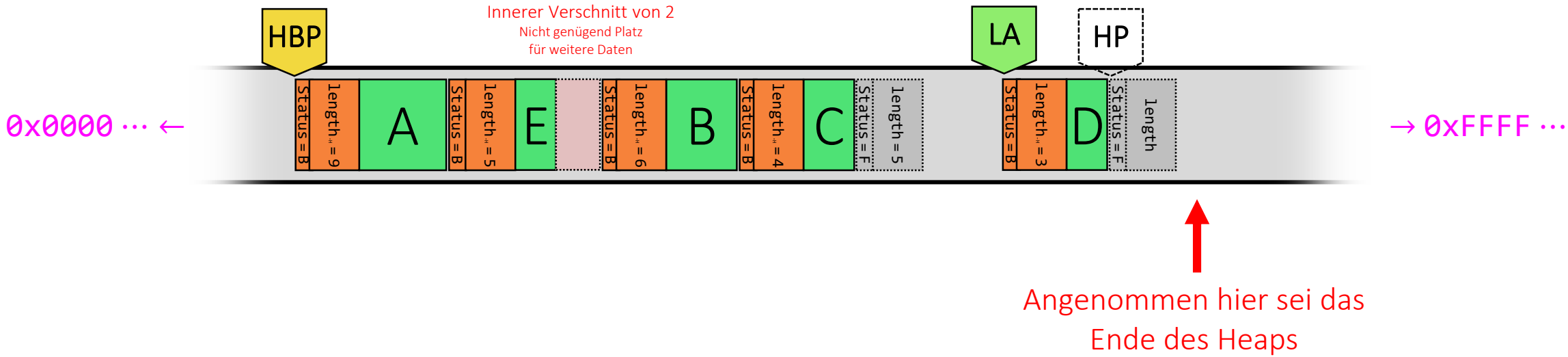
- Suche nach einem freien Bereich
  - Falls Bereich reserviert, um Anzahl Blöcke weiter springen
  - Wenn Ende des Speichers erreicht, dann Vorne beginnen

# Allokation eines Blockes $\mathbf{E}$ der Länge $\mathbf{3}$



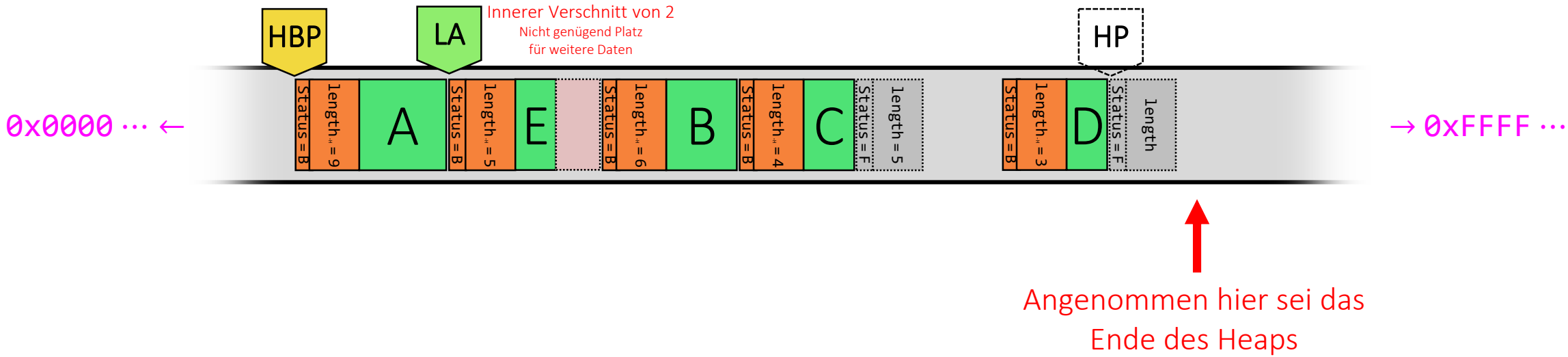

**H** entspricht hier der Länge der Metadaten.

# Allokation eines Blockes E der Länge 3




**H** entspricht hier der Länge der Metadaten.

# Allokation eines Blockes E der Länge 3

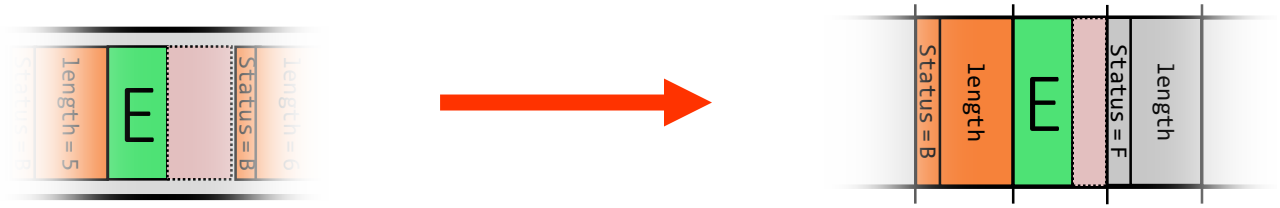



**H** entspricht hier der Länge der Metadaten.

# Chunking

- Heap wird in sehr kleine, gleichgroße Blöcke (**Chunks**) geteilt
- Bei der Reservierung von Speicherbereichen wird die geforderte Speichergröße in Chunks umgerechnet und aufgerundet

  → Ggf. innerer Verschnitt
- Chunkgröße ≥ Metadaten ⇒ jede freie Lücke im Speicher kann auch als solche entsprechend markiert werden: In der Implementierung spart man sich so eine Fallunterscheidung!



# Speicherplatz freigeben

- Gegenstück zu `malloc` (bzw. `calloc`) ist `free(void* ptr)`
- Gibt Speicher an Adresse `ptr` wieder frei
- Speicher darf nur einmal freigegeben werden
- `free` bedarf keine Fehlerbehandlung

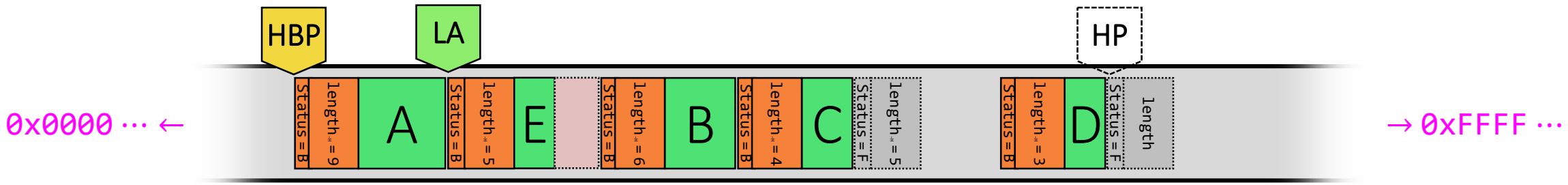
# Verwendung von `free`

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int n = 200;

int main(void) {
    int* ptr;
    ptr = malloc(n * sizeof(*ptr));
    if (ptr == NULL) { perror("malloc"); exit(125); }
    for (int i = 0; i < n; ++i) { ptr[i] = i * i; }
    printf("10*10 = %d\n", ptr[10]);
    free(ptr);
      ⋮
    return 0;
}
```

# Freigabe

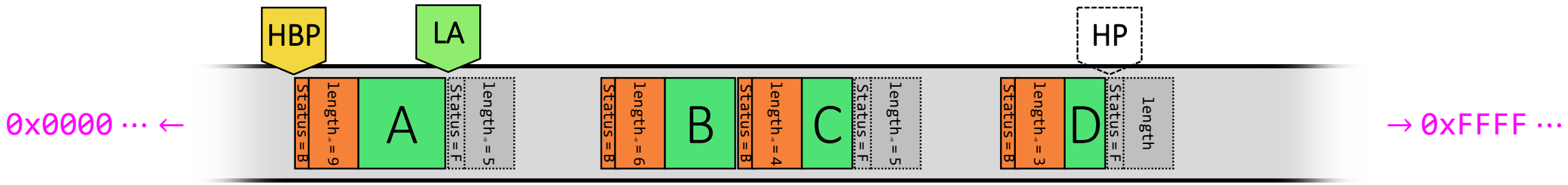
- Markiere Block als frei
- Angrenzende Bereich überprüfen
  - Bereich davor frei? → Verschmelzen
  - Bereich dahinter frei? → Verschmelzen

# Vor der Freigabe von E und C



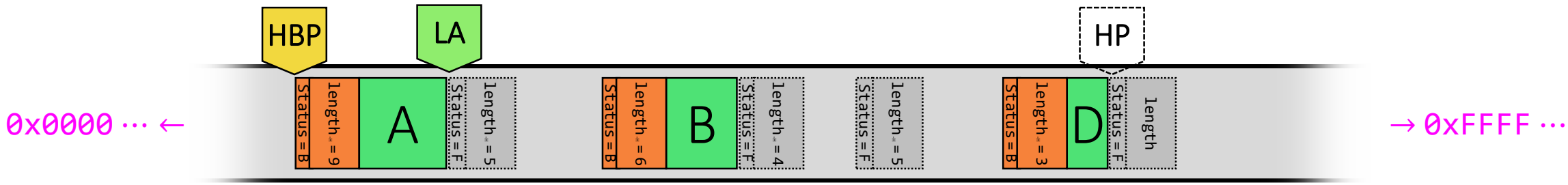

H entspricht hier der Länge der Metadaten.

# Nach der Freigabe von E



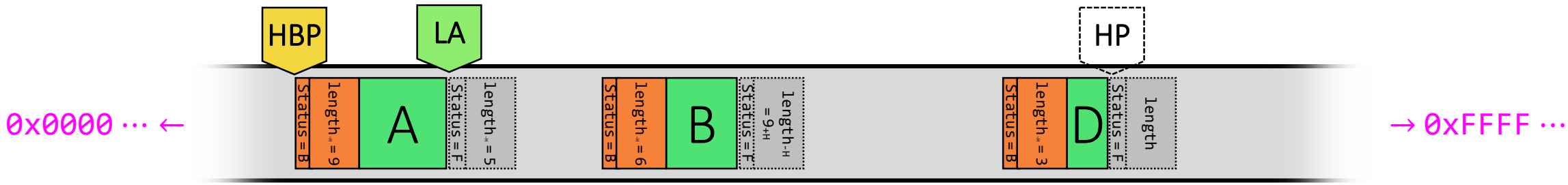

**H** entspricht hier der Länge der Metadaten.

# Nach der Freigabe von C




**H** entspricht hier der Länge der Metadaten.

# Verschmelzen




H entspricht hier der Länge der Metadaten.

# Speicherplatzierungsstrategie 'Best Fit'

- Sucht die am besten passende Lücke im Speicher
- Aufwendigere Platzierungsstrategie die aber für größere Blöcke sehr gute Ergebnisse liefert